Sem vložte zadání Vaší práce.

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra teoretické informatiky

Bakalářská práce

Implementace evolučního shlukování

Jan Trusina

Vedoucí práce: Ing. Tomáš Bartoň

13. února 2015

Poděkování

Tímto bych chtěl poděkovat panu Ing. Tomáši Bartoňovi za vedení mé bakalářské práce, cenné rady a obětavou pomoc. Také bych zde chtěl poděkovat rodičům a celé rodině za podporu během dosavadního studia a pochopení občasných neúspěchů.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 13. února 2015 .....

Odkaz na tuto práci

Trusina, Jan. *Implementace evolučního shlukování*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2015.

Abstrakt

Tato bakalářská práce se zabývá shlukováním dat a implementací algoritmu MOCK. První část této práce je věnována různým metodám shlukování dat. V druhé části je popsán algoritmus MOCK a jeho implementace do frameworku Clueminer. Dále pak popis nasazení a výsledků algoritmu na reálných datech.

Klíčová slova evoluční shlukování, MOCK, Clueminer, shluk

Abstract

This bachelor thesis deals with cluster analysis and implemenatiton of MOCK algorithm. The first part of this thesis acquaints with the principles of clustering. The second part describes algorithm MOCK and its implementation to the Clueminer framework. Then the description of deployment on real datasets is presented.

Keywords evolutionary clustering, MOCK, Clueminer, cluster

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Shlukování dat je metoda sloužící ke třídění vícerozměrných dat. Shluk je skupina dat, která je v rámci svého shluku navzájem maximálně podobná a současně proti ostatním shlukům maximálně odlišná. Tato metoda je stavebním kamenem pro vytěžování znalostí dat a v dnešní době se využívá např. v marketingu k analyzování chování spotřebitelů, v biomedicínské informatice k zjištění vzorů nemocí či v mnoha dalších odvětvích.

Shluková analýza není žádný specifický algoritmus, jedná se o obecnou úlohu. Existující metody rozdělujeme na hierarchické, dělící, hustotové, mřížkové, konceptuální, metody neuronových sítí či evoluční metodu. Každá z těchto metod má své využití a také výhody a své nevýhody, které vám autor přiblíží v první části této práce. Kromě výše zmíněných metod, výsledek zásadně ovlivňuje i to, jaká metrika se použiívá, nejčastěji je to metrika euklidovská [1], ale legitimní jsou i jiné metriky, vhodný výběr záleží také na povaze dat.

Evoluční algoritmy se využívají k prozkoumání velkého stavového prostoru, podobně jako v biologii i u evolučních algoritmů dochází ke křížení, mutaci apod. Nejprve však algoritmus v inicializační části připraví vstupní sadu řešení pro daná data.

Poté algoritmus přejde do evoluční části, kde postupně vytváří nové generace, které se vyhodnocují, a jedince, kteří prokáží dostatečné zlepšení, vymění za stará a horší řešení, která zahodí. Tento postup iteruje předem nastavených počtem generací. Nakonec se vyhodnotí a podle hodnotící funkce nejlepší řešení.

Autorovou osobní motivací, proč si vybral toto téma, byl fakt, že shledával problematiku analyzování dat vždy velice zajímavou. Zajímá ho problematika data miningu i jako možné pole pro podnikání po dostudování školy.

Cíl práce

Cílem této bakalářské práce je seznámení se s metodami shlukování, algoritmem MOCK, jeho implementací do frameworku Clueminer a otestováním na datech. Výsledky algoritmu budou porovnány s výsledky jiných, tradičních, algoritmů na pseudonáhodných datech, ovšem tyto data mají nevýhodu právě kvůli tomu, že byla vygenerována náhodně, tudíž je téměř nemožné dosáhnout takových výsledků, jako algoritmy založené na hustotě, které těží právě z pseudonáhodnosti dat. Naopak pro velké datasety, kde by výpočty heuristický algoritmů probíhaly dlouho, je vysoká pravděpodobnost, že evoluční algoritmy naleznou poměrně lehce velmi dobré řešení. Ve své práci však autor bude testovat data obou typů.

Struktura dokumentu

První kapitola přiblíží problematiku shlukování dat, popíše nejznámější a nejpoužívanější techniky, které se používají v dnešní době.

Druhá kapitola rozebírá algoritmus MOCK[4], tento algoritmus je stěžejním bodem této práce. Autor vám přiblíží, jak algoritmus funguje.

Třetí kapitola popisuje framework Clueminer.

Čtvrtá kapitola se věnuje analýze problému a návrhu implementace. V této kapitole je popsáno, jak byl algoritmus MOCK implementován.

Pátá kapitola přiblíží, jak probíhalo testování implementace. Také se dozvíte, jaké výsledky měla různá data v porovnání s výsledky tradičních programů.

V závěru autor zhodnotí svou práci, algoritmus a výsledky, které získal v testování algoritmů.

Kapitola 1

Techniky shlukování dat

Shlukování je obecná úloha, jejíž průběh a výsledky velmi ovlivňuje několik faktorů ve všech stádiích algoritmu. Záleží například na vstupních datech, protože většinou nejsou známy dodatečné vlastnosti jednotlivých tříd, tudíž nelze přiřadit prioritu k jednotlivým atributům, proto občas shlukování bývá označováno jako optimalizace více funkcí.[1] V závislosti na povaze dat se nasazují různé algoritmy. Tyto algoritmy se dělí primárně hierarchické a nehierarchické. Hierarchické metody charakterizuje to, že jejich shluky nejsou disjunktní, tedy jejich průnik nemusí vždy tvořit prázdná množina. Hierarchické metody se dělí na dva typy přístupu, aglomerativní a divizní.

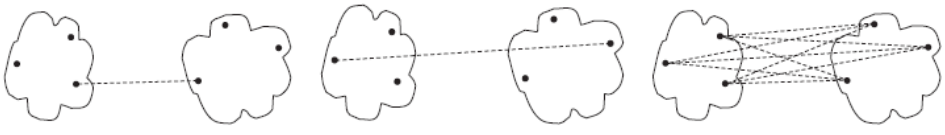
1.1 Hierarchické metody

Aglomerativní přístup je klasický příklad strategie bottom up, tedy kdy, jak už napovídá anglické označení, strom se tvoří zespodu. Na začátku tvoří každý uzel svůj vlastní shluk a postupně se slučují shluky k sobě tak dlouho, dokud se nezredukuje počet shluků na předepsanou hodnotu.

Naopak divizivní přístup je klasický příklad strategie top down, tedy na začátku jsou všechny vrcholy v jednom shluku a postupně shluk dělím tak dlouho, dokud se nezíská požadovaný počet shluků. K výpočtům se používá pomocná matici vzdáleností, do které se předpočítají všechny vzdálenosti, aby se hodnoty nemusely počítat pořád znovu. Je důležité, jaká se zvolí metrika vzdálenosti mezi shluky, resp. s jakou hodnotou (pozicí) centra shluku se počítá. Na výběr jsou tyto typy:

- metoda nejvzdálenějšího souseda (*complete-linkage*)
- metoda nejbližšího souseda (*single-linkage*)
- centroidní metoda (*average-linkage*)

[1] angl. multiobjective optimization



(a) MIN (single link.) (b) MAX (complete link.) (c) Group average.

Obrázek 1.1: Schéma nejbližší / nejvzdálenější soused, centroidní metoda; zdroj: [6]

- Wardova metoda (Ward's method[5])
- a mnoho dalších ...

Metoda nejvzdálenějšího souseda je jedna z aglomerativních hierarchických metod. Algoritmus běží buď předepsaný počet cyklů nebo dokud nevytvoří jeden jediný cluster (strom). Na začátku tvoří každý prvek svůj vlastní shluk, v každé iteraci se spojí dva nejbližší shluky, v případě této metody se vzdálenost počítá pomocí nejvzdálenějších prvků v clusteru. Naivní algoritmus pro tuto metodu je triviální, ovšem asymptotická složitost dosahuje $\mathcal{O}(n^3)$, v roce 1977 tuto metodu zoptimalizoval D. Defays na $\mathcal{O}(n^2)$, algoritmus se jmenuje *CLINK* [7], tento algoritmus je obdobný jako algoritmus *SLINK* [8] pro metodu nejbližšího souseda.

Metoda nejbližšího souseda je další z aglomerativních hierarchických metod. Algoritmus běží na podobném principu jako metoda nejvzdálenějšího souseda avšak používá pro měření vzdáleností nejbližší prvek ve shluku. Podobně jako u metody nejvzdálenějšího souseda i v této metodě naivní algoritmus dosahuje složitosti $\mathcal{O}(n^3)$ a byl zoptimalizovaný v roce 1973 R. Sibsonem na $\mathcal{O}(n^2)$, algoritmus pojmenoval *SLINK* a dále ho upravil i D. Defays pro metodu nejvzdálenějšího souseda. Tento algoritmus funguje podobně jako Kruskalův algoritmus[9] pro vytvoření minimální kostry grafu a i autor ho používá ve své implementaci pro počáteční inicializaci.

Centroidní metoda je další z aglomerativních metod. Od předchozích dvou metod se liší hlavně v tom, že svůj výchozí bod pro počítání vzdáleností má ve svém aritmetickém středu, tudíž velmi pravděpodobně mimo uzel shluku, pokud shluk netvoří pouze jeden shluk. Logicky se pro tuto metodu nabízí používat euklidovskou metriku. Tento algoritmus se používá pod názvem *UPGMA*.[2] I tento algoritmus dosahuje ve své optimalizované implementaci $\mathcal{O}(n^2)$.

Wardova metoda je poslední z aglomerativních metod, která zde je popsána. Tato metoda funguje na principech analýzy rozptylu a slučuje takové

[2]angl. *Unweighted Pair Group Method with Arithmetic Mean*,[10]

shluky, které mají minimální součet čtverců. Také tato metoda vyžaduje euklidovskou metriku pro počítání vzdáleností.

1.2 Dělící metody

Dělící metody mají za cíl rozdělit všechny prvky na předem daný počet shluků. Shluky dělí podle podobnostních funkcí. Je zde na výběr, zda se použije pro centrum shluku těžiště – k-Means (obdoba centroidní metody) nebo vrchol nejblíže těžišti – K-Medoids[11]. Dělící algoritmy jsou vhodné pro malý počet shluků z velkého počtu objektů a nejsou příliš odolná na špatná data, kde je hodně nekonzistencí a šumů. Tuto metodu využívá například algoritmus CLARANS [12], který se využívá pro tzv. spatial data mining.

1.3 Hustotové a další metody

Metody založené na hustotě využívají prostorovou hustotu objektů. Metody se řídí pomocí dvou parametrů, maximálního rádia okolí a minimálního počtu prvků v okolí. Aby byl prvek jádrem, musí splňovat, že ve vzdálenosti rádia musí mít alespoň minimální počet prvků. Nejčastěji používaným algoritmem je DBSCAN [13].

Dalšími metodami jsou ještě např. mřížkové metody, které mají za úkol rozdělit prvky pomocí mřížek (šachovnice), nebo metody neuronových sítí. Metod je ovšem ještě daleko více, avšak protože tato práce není jen o shlukové analýze, dle autora tolik bude stačit k přiblížení problematiky.

1.4 Požadavky na shlukovací metody

Abychom dosáhli správných výsledků, námi zvolená metoda by měla umět vyhovět většině požadavků níže zmíněných.

- zpracování mnoha dimenzionálních dat
- zpracování různých typů dat
- zpracování rozsáhlých dat
- nalezení ideálního počtu shluků bez znalosti dodatečných informací o datech
- interpretace výsledků

Kapitola 2

Algoritmus MOCK

Algoritmus MOCK (multiobjective clustering with automatic k-determination) [4] je algoritmus pro shlukování s optimalizací více funkcí. Skládá se ze dvou částí. V první, shlukovací části algoritmus pojmenovaný PESA-II [14], používá evoluční algoritmus uzpůsobený pro práci s více-dimenzionálními daty. V této části probíhá evoluce na datech, tedy pro cca tisíc iterací zkouší mutovat, křížit apod. jednotlivé individuály. Výstupem této části je množina různých řešení, takových, jak je evoluce změnila. Druhá část se nazývá modelová a vybírají se z ní nejlepší řešení z první části. MOCK změří tvar zakřivení křivek u všech řešení a porovná ho s modelem pro náhodná data. Pomocí měření zjistí algoritmus přibližnou kvalitu jednotlivých řešení a vybere z nich množinu nejlepších (Paretovsky optimálních) řešení.

> Optimalizace více funkcí je používána pro vypořádání se úkolu učení bez učitele, shlukování dat. Konceptuální výhody optimalizace více funkcí jsou diskutovány a evoluční přístup k problému je vyřešen. Výsledný algoritmus, MOCK (Clusterovací algoritmus optimalizace více funkcí s automatickou k-determinací), je porovnán s výsledky algoritmů fungujících pro jednoduchá data. Experiment demonstruje, že konceptuální výhody shlukování podle více funkcí použité v praxi, jsou výhodné. [4]

2.1 Shlukovací část

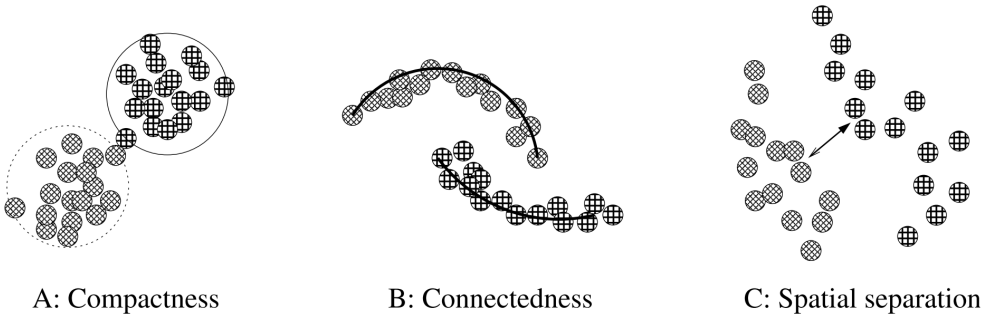
Algoritmus si průběžně udržuje 2 populace, interní a externí. Interní populace má omezenou velikost, externí nemá fixní velikost ale pouze maximální velikost. Úkolem externí populace je nacházet dobrá průběžná řešení. Využívá k tomu princip elitismu. Úkolem interní populace je nacházet nová řešení pomocí klasických procesů evolučních algoritmů, klonování, rekombinace a mutace.

Shlukovací část obsahuje několik operací, popsány zde jsou postupně tyto části:

- Hodnotící funkce
- Funkce odchylky
- Funkce propojení
- Grafová reprezentace
- *Control* fronta
- Rovnoměrné křížení
- Mutace
- Inicializace
- Iterace evoluce

2.1.1 Hodnotící funkce

Handl a Knowles nabízejí 3 funkce na hodnocení kvality řešení. Ačkoli nabízejí



Obrázek 2.1: Kompaktnost, propojení a prostorové oddělení.

funkce kompaktnosti, propojení a prostorového oddělení, doporučují používat pouze první dvě, protože funkce propojení a prostorového oddělení jsou založeny na podobném konceptu, na rozdíl od funkcí kompaktnosti a propojení, které očekávají od vlastností shluku jiné parametry, a tudíž je lepší vybrat právě tyto dvě funkce. Obě mají výborné předpoklady pro ovládání se navzájem ve zvětšování či snižování počtu clusterů. Zatímco odchylka se zlepšuje spolu s vyšším počtem clusterů, u propojenosti tomu tak je naopak, čím méně clusterů, tím lepší propojenost.

2.1.2 Funkce odchylky

Funkce odchylky (*overall deviation*) počítá kompaktnost pomocí rozdílu vzdáleností mezi prvky a jejich příslušnými středy shluků.

$$Dev(C) = \sum_{C_k \in C} \sum_{i \in C_k} \delta(i, \mu_k), \tag{2.1}$$

kde C je množina všech shluků, μ_k je střed shluku C_k a $\delta(.,.)$ je zvolená vzdáleností funkce. Naším úkolem je tuto odchylku co nejvíce minimalizovat.

2.1.3 Funkce propojení

Propojovací funkce se počítá pomocí propojenosti, která počítá, kolik z nejbližších L prvků je ve stejném shluku.

$$Conn(C) = \sum_{i=1}^{N} (\sum_{j=1}^{L} x_{i,nn_{ij}}) \tag{2.2}$$

$$\text{kde } x_{r,s} = \begin{cases} \frac{1}{j} & \text{když } \nexists C_k : r \in C_k \wedge s \ \in C_k \\ 0 & \text{jinak,} \end{cases}$$

kde $nn_{i,j}$ je j-tý nejbližší soused uzlu i, N je velikost clusterovaného datasetu, L je parametr nastavený při inicializaci programu, který určuje, kolik nejbližších sousedů se bude hledat. Obdobně jako u propojovací funkce i zde je cílem, aby propojenost byla co nejmenší.

2.1.4 Control fronta

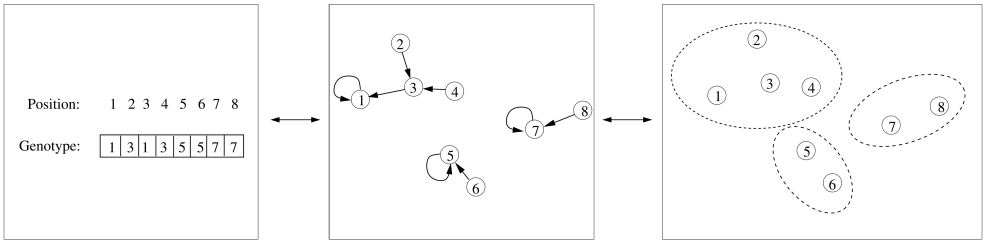
Tato fronta se slouží k porovnání s výslednými řešeními. Odhaduje výsledky měřících funkcí pro nestrukturovaná data, tedy je potřeba vygenerovat za pomoci PCA[3] vlastní vektory.[4] Pomocí vektorů se pak následně vygenerují data, která se ještě musí normalizovat do intervalu hodnot původního datasetu. Na tento nový dataset se pak pustí znovu inicializační metoda, která nalezne počáteční řešení a změří se hodnoty hodnotících funkcí *connectivity* a *deviation*, které budou potřeba pro porovnání s výslednou *Pareto* frontou.

2.1.5 Grafová reprezentace

Hrany mezi jednotlivými prvky jsou reprezentovány N geny. Každý gen představuje potenciální hranu z prvku. Ačkoli na obrázku pod textem je graf orientovaný, ve skutečnosti se jedná o neorientované grafy. Ukazují pouze z jakého vrcholu do jakého vede hrana, ale jedná se jen o neorientovanou hranu. Mezi

[3] angl. Principal Component Analysis, Analýza hlavních komponent [15]
[4] angl. Eigenvalues a Eigenvectors

dvěma vrcholy může vést maximálně 1 přímá hrana. Na tuto reprezentaci je potřeba pamatovat při implementaci, aby se správně generovaly shluky.



Obrázek 2.2: Grafová reprezentace.

Výhodou této reprezentace je, že se na ni dobře aplikuje během evoluce současně více způsobů (křížení apod.) Protože při vícenásobném křížení může dojít k velkým změnám ve struktuře, tak se přepočítávají shluky až na konci každé iterace (generace). Úprava shluků po každém zásahu do struktury by byla sice momentálně jednodušší, protože by se např. při výměně jedné hrany změna dotýkala jen max. 2 shluků, ale při hlubšímu zásahu je už jednodušší všechny shluky znova vygenerovat.

2.1.6 Vícenásobné křížení

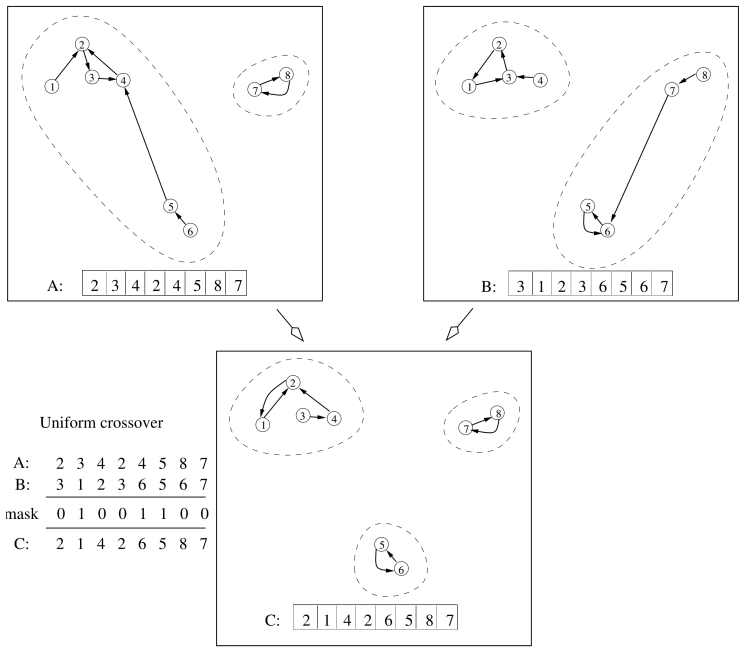
Vícenásobné křížení[5] má na rozdíl od rovnoměrného křížení výhodu v nepředpojatosti k pořadí genů, a může vytvořit jakoukoli kombinaci ze dvou rodičů. Handl a spol. přesně nepopisují, jestli je vhodné přidávat nějakému rodiči vyšší pravděpodobnost, proto lze vycházet z předpokladu, že každý gen je zkřížen s pravděpodobností 50%.

2.1.7 Mutace

V případě mutace[6] využívá algoritmus PESA-II omezení, aby nedocházelo k procházení zbytečně velkého prostoru, když stačí teoreticky pouze předem daný maximální počet L nejbližších sousedů, se kterými lze křížit. Samozřejmě si list nejbližších sousedů algoritmus předpočítá na začátku inicializace, aby nemusel počítat v každé iteraci už spočítané vzdálenosti. Pro L obvykle platí $L \ll N$. Protože algoritmus chce preferovat kratší hrany, může použít předsudkovou rovnici:

$$p_m = \frac{1}{n} + \left(\frac{l}{N}\right)^2, \tag{2.3}$$

[5] angl. uniform crossove
[6] angl. neighbourhood-biased mutation operator



Obrázek 2.3: Vícenásobné křížení.

kde l je l-tý nejbližší prvek. Logicky také musí platit: $l \leq L$. Tato rovnice lehce nalezne nevýhodné hrany.

2.1.8 Inicializace

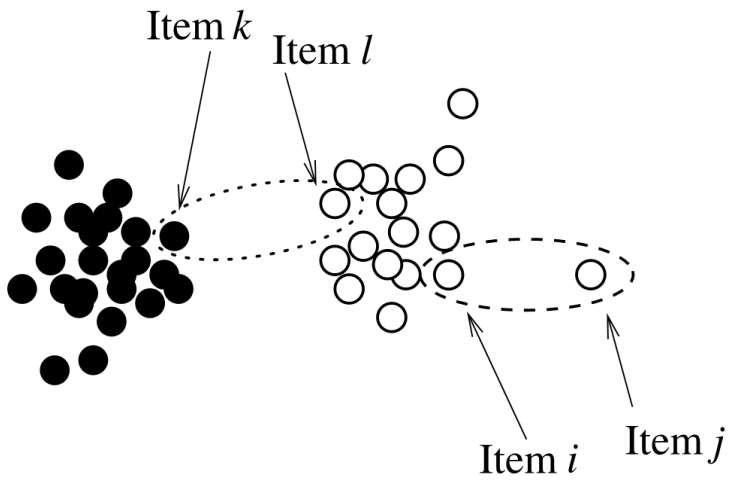
Použije-li algoritmus pro počáteční rozdělení hran Primův algoritmus [16] pro minimální kostru grafu (minimal spanning trees – MSTs), zaručí to dobrou propojenost. Vygenerovaná kostra je již přímo použitelná pro evoluční algoritmus. Pro dobrou (minimální) odchylku prvotních generovaných hran se používá algoritmus k-Means.

Generování zajímavých MST řešení

PESA-II pro svou inicializaci používá taktéž Primův algoritmus, tedy MST. Aby se vyvaroval pouze odebírání nejdelších hran, které by pouze izolovaly vychýlené prvky (tzv. outliers) a řešení by byla velmi podobná, vytvoří koeficient zajímavosti, který odlišuje hrany na zajímavé a nezajímavé podle toho, zda po odebrání získá outlier nebo celý cluster. Hrana $i \rightarrow l$ je považována za zajímavou, pokud

$$i = nn_{jl} \wedge j = nn_{ik} \wedge l > L \wedge k > L \tag{2.4}$$

Shlukové řešení C je považováno za zajímavé, pokud může být vytvořeno pouze odebráním zajímavých hran z minimální kostry.



Obrázek 2.4: Zajímavá hrana a outlier

Set zajímavých MST se vytvoří následujícím způsobem. Nejdříve se spočítá počet zajímavých hran a seřadí se podle stupně zajímavosti, následně se vytvoří set shlukových řešení pro

$$n \in 0, \ldots, min(I, \lfloor 0.5 * fsize \rfloor - 1), \tag{2.5}$$

kde I je počet zajímavých hran, $fsize$ je celkový počet inicializačních řešení, řešení C_n je vygenerováno odebráním prvních n zajímavých hran. Hrana odebraná na pozici i je nahrazena hranou do náhodně vybraného souseda j.

$$j = nn_{il} \wedge l \leq L \tag{2.6}$$

Generování zajímavých *k*-Means řešení

Druhou částí inicializace je generování k-Means řešení. Algoritmus nechá běžet pro k-Means po 10 iterací pro různé velikosti clusterů

$$k \in \{2, \ldots, fsize - min(I, \lfloor 0.5 * fsize \rfloor - 1)\}, \tag{2.7}$$

je zřejmé, že počet zajímavých MST řešení (ze vzorce 2.5) a počet k-Means (ze vzorce 2.7) dosahují přesně hodnoty $fsize - 1$.

2.1.9 Iterace evoluce

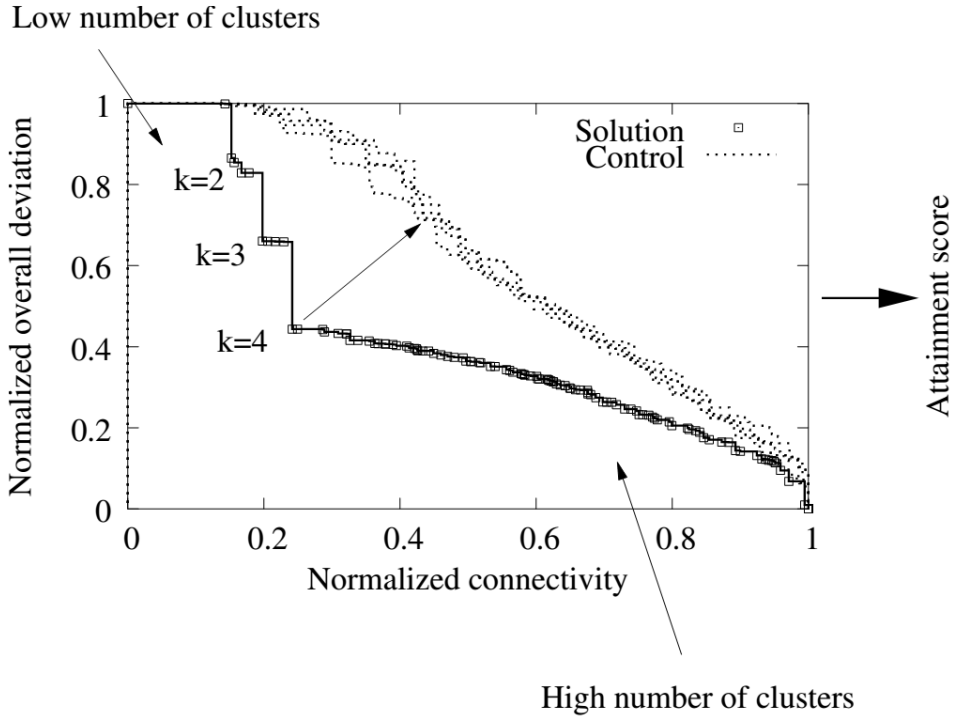
Smyčka programu se sem dostane po dokončení inicializace a naplnění externí populace výsledky z generování MST a k-Means. Algoritmus PESA-II zde běží předem nastavený počet iterací a probíhá zde mutace a křížení. Ve chvíli, kdy doběhne smyčka, končí clusterovací část algoritmu a následuje část modelová.

2.2 Modelová část

Shlukovací část dodá do modelové části externí populaci, což je množina možných řešení. Jednotlivá řešení se liší rozdíly v naplnění dvou úkolů, odchylky a propojenosti, ale také v rozdílném počtu clusterů. Algoritmus MOCK se nyní pokusí najít řešení, které je co nejvíce Pareto optimální pomocí poměrně jednoduchého algoritmu. Nejdříve odebere všechna řešení, která mají více clusterů, než si uživatel nastavil na začátku. Následně zjistí minimální a maximální hodnoty odchylky a propojenosti, které použije pro normalizaci. Normalizace je nastavena na 0 pro nejmenší hodnotu a 1 pro největší, tedy odpovídající funkce:

$$\frac{x-a}{b-a}, \tag{2.8}$$

kde x je hodnota, která se normalizuje, a je nejmenší nalezená hodnota a b je největší hodnota. Nyní už je potřeba pouze nalézt nejlepší řešení, které algoritmus nalezne pomocí tzv. dovednostního skóre (attainment score).



Obrázek 2.5: Hledání nejlepšího řešení

Dovednostní skóre se počítá pro každé výsledné řešení, cílem je nalézt dominující řešení, jehož výsledek je co největší z minimálních euklidovských rozdílů ostatních řešení a nejbližšího řešení *Control* fronty.

2.3 Nastavení parametrů

Správné nastavení parametrů podstatně ovlivňuje správnou funkcionalitu algoritmu. Handl a Knowles speciálně upravují tyto parametry:

- Velikost externí populace
- Počet generací
- Velikost interní populace
- Rozlišení mřížky na dimenzi
- Maximální počet shluků k_{user}
- Počet inicializačních řešení f_{size}
- Inicializace
- Typ mutace
- Četnost mutace p_m
- Typ rekombinace
- Četnost rekombinace p_c
- Hodnotící funkce
- Proměnná L

Bylo zjištěno, že maximální velikost 1000 prvků externí populace je dostatečná pro to, aby dokázala udržet všechny podstatná nedominující řešení. Při větším počtu prvků externí generace se rychle zvyšuje paměťová i časová náročnost programu a přidaná hodnota v počtu prvků populace není dostatečně velká, aby se vyplatilo na ní cílit na úkor složitostí.

Pro počet generací zvolil Knowles a spol. také hodnotu 1000 s tím, že je to dostatečně velká hodnota, zaručující dostatečnou mutaci. Je samozřejmé, že pro vyšší čísla by měla být nalezena kvalitnější řešení, ale algoritmus je již takto náročný a tyto hodnoty byly nalezeny a experimentálně ověřeny jako dostatečné v porovnání se správností výsledku.

Rozlišení mřížky bylo stanoveno na hodnotu 10. Tedy v případě této práce při použití 2 hodnotících funkcí vznikne pole 10x10. Na první pohled se může zdát, že takové rozšíření je příliš malé, bylo by lepší například použít 100x100

pro větší přesnost a pracování pouze s nejlepšími výsledky. Ovšem rozlišení je nastaveno velice správně na takovou hodnotu, aby se zbytečně nezahazovala řešení, která mohou být krátkodobě horší, ale mají potenciál se stát lepší v průběhu další evoluce.

Pro nastavení maximálního počtu shluků k_{user} autoři doporučují hodnotu 25 pro malé datasety a 50 pro ostatní. Počet inicializačních řešení f_{size} je dvojnásobek maximálního počtu clusterů, tedy $2 \cdot k_{user}$ Při inicializaci používají minimální kostry grafů a k-Means.

Pro typ mutace se používá L nejbližších sousedů. Proměnná L společná proměnná pro více parametrů programu a bude popsaná níže. Pro nastavení pravděpodobnosti mutace se používá L nejbližších sousedů a každý gen se mutuje s pravděpodobností

$$p_m = \frac{1}{n} + \left(\frac{l}{N}\right)^2. \tag{2.9}$$

Křížení probíhá s pravděpodobností $p_c = 0.7$ za použití vícenásobného křížení. Proměnná L, jak již bylo zmíněno výše, je proměnnou pro více parametrů programu. Je použita v MST jako omezení L nejbližších vrcholů, které definují, jestli se jedná o outlier nebo o zajímavou hranu. Dále pak u k-Means jako nastavení maximálního počtu iterací při hledání center clusterů, potom je také tato proměnná použita u mutace (podobně jako u MST) pro omezení počtu nejbližších sousedů. Hodnota proměnné L je nastavena na hodnotu 10, doporučený interval pro výběr hodnoty je $L \in \{5, \ldots, 20\}$.

Hodnotící funkce získaných řešení jsou funkce odchylky a propojenosti. Toto nastavení zásadně ovlivňuje správnou funkcionalitu programu, avšak to neznamená, že za použití jiných nastavení by se algoritmus nedostal ke správnému výsledku. Algoritmus implicitně používá doporučené nastavení, o jeho drobných úpravách se můžete dočíst v kapitole Popis metodiky testování a jeho výsledky (viz. kapitola 5).

Kapitola 3

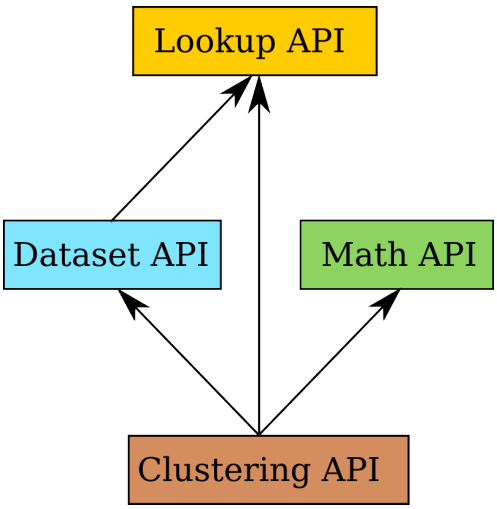
Framework Clueminer

Clueminer je nástroj pro explorační datovou analýzu, který byl vytvořen v rámci magisterské práce [17]. Na rozdíl od ostatních frameworků pro data-mining (RapidMiner, ELKI, Orange) Clueminer používá modulární architekturu a umožňuje jednoduché rozšíření pomocí samostatného modulu.

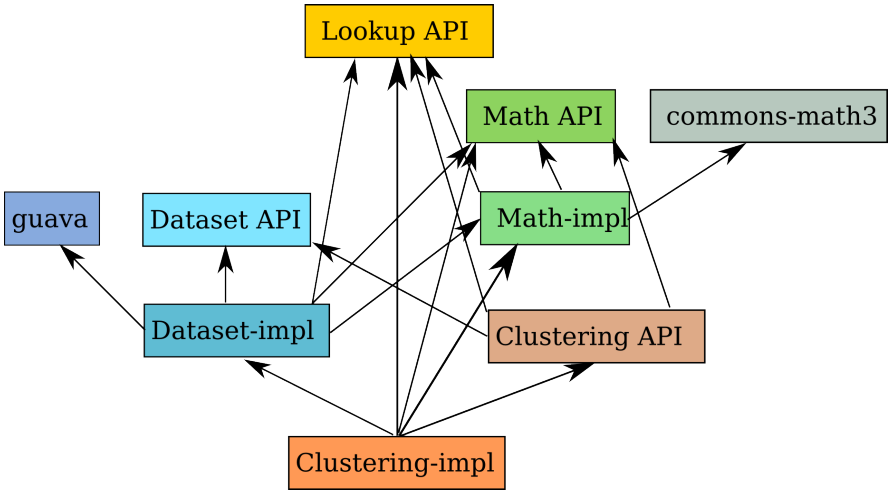
Clueminer je postaven na NetBeans Plaform[7], což je platforma použitá mimo jiném pro IDE NetBeans. Z hlediska návrhu software se jedná o aplikování principu *loose coupling and high cohesion* [18]. Tedy jednotlivé třídy by neměly záviset na mnoha třídách z ostatních balíků, ale využívat především třídy ze stejného balíku.

V rámci frameworku Clueminer je důležité oddělení algoritmů od grafické

[7] `https://netbeans.org/features/platform/`



Obrázek 3.1: Základní moduly platformy Clueminer. Moduly definují pouze API, ale implementace je v modulu, který je závislý na tomto API.



Obrázek 3.2: Implementace základních API v rámci platformy Clueminer. V rámci architektury modulů není možné vytvářet cyklické závislosti. Modul, který používá `Clustering-impl` nemusí mít deklarovanou závislost na tento modul, stačí pouze `Clustering-api`, konkrétní implementaci je možné načíst pomocí `Lookup`.

(prezentační) vrstvy. Grafické rozhraní závisí na API, které definuje přístup k datů (*Dataset API*) a definici rozhraní algoritmů pro shlukovou analýzu (*Clustering API*). Tento návrh umožňuje vytvoření minimální aplikace, která obsahuje pouze příkazovou řádku a definice algoritmů. Taková aplikace je vhodná např. pro spouštění analýzy na výpočetních serverech. Zároveň stejné module je možné využívat grafickém rozhraní, které vizualizuje výsledky shlukové analýzy.

Díky tomuto návrhu je možné jednoduše nahradit modul, který se stará o interní reprezentaci dat a nahradit jej jiným modulem, aniž bychom musel refaktorovat další kód.

3.1 Lookup

Lookup je jednou ze základních komponent NetBeans, která umožňuje získat instance určité třídy. Následující kód najde všechny instance, které implementují požadované rozhraní.

```
Collection list = Lookup.getDefault().lookupAll(Clustering.class);
```

Modul třetí strany implementující dané rozhraní může jednoduše rozšířit funkce aplikace bez nutnosti zásahu do původních zdrojových kódů.

Tento mechanismus se používá např. pro implementaci metriky vzdálenosti. Všechny metriky vzdáleností jsou definované v jednom modulu (*high*

cohesion), nicméně modul třetí strany může definovat novou metriku, která se díky stejnému API a definici `ServiceProvider` může využívat v rámci celé aplikace.

```
@ServiceProvider(service = DistanceMeasure.class)
public class EuclideanDistance implements DistanceMeasure
  // ...
}
```

Anotace `@ServiceProvider` zajišťuje registraci třídy po přidání třídy na `CLASSPATH` Java aplikace.

3.2 Dataset API

Rozhraní Dataset API definuje přístup ke standardním „tabulkovým“ datům (tj. nejedná se např. o grafové struktury). V rámci takového datasetu se používá konvence z oblasti data-miningu, kdy jednotlivé atributy (vstupy, proměnné) jsou uvedené ve sloupcích a pozorování (měření) jsou v řádcích. Každý takový řádek může mít přiřazenou třídu (label), což se typicky používá v úlohách pro učení s učitelem, nicméně v rámci Cluemineru je informace o třídě použita pro vyhodnocení výsledků shlukové analýzy. Pro „sloupce“ (atributy) jsou automaticky počítány statistiky, jelikož je to informace, kterou potřebuje řada algoritmů.

Modul `dataset-impl` poskytuje několik implementací rozhraní Dataset API, které je postavené na rozhraní Java Collections. Nicméně standardní kolekce jako `ArrayList` se z důvodu optimalizace pro výkon příliš nevyužívají.

3.3 Clustering API

Každý shlukovací algoritmus by měl implementovat rozhraní z tohoto modulu. Rozhraní shlukovacích algoritmů předpokládá dataset definovaný v Dataset API na vstupu a výstupem je třída `Clustering`, což je kolekce shluků, které obsahují reference na řádky datasetu.

Díky obecnému rozhraní pro jednotlivé shlukovací algoritmy je pak možné psát meta-algoritmy, které pracují na kombinaci více algoritmů, aniž by musely předem vědět o jejich existenci.

Kapitola 4

Analýza a návrh implementace

V této kapitole je popsáno, jak probíhala analýza a implementace. Analýza problému byla poměrně jednoduchá, protože pro se autor musí držet poměrně striktního zadání algoritmu a je značně omezen i frameworkem, do kterého musí algoritmus implementovat. Framework Clueminer je napsaný v Javě, tudíž i autor implementoval algoritmus v tomto jazyce. Dále je zde patrná snaha co nejvíce využívat možností frameworku i při vlastní implementaci, například euklidovskou vzdálenostní funkci pro metriku atd. Pro úložiště grafů byl použit balíček GraphStore[8] z repositáře Gephi[9].Z důvodu komplexnosti samotného Gephi je však pro vizualizaci grafů použit jednodušší JFreeChart[10].

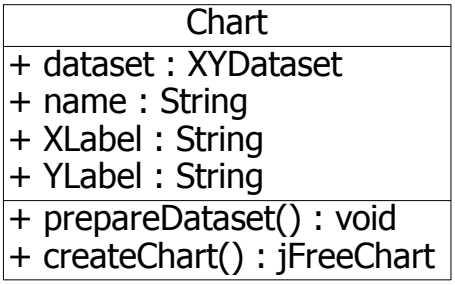
Samotná implementace algoritmu MOCK obsahuje následující třídy:

- Chart
- Individual
- MST
- Matrix Of Lengths
- Mock
- Node Instance
- kMeans

[8] https://github.com/gephi/graphstore
[9] https://gephi.github.io/
[10] http://www.jfree.org/jfreechart/

4.1 Třída Chart

Tato třída slouží k vykreslování grafů. Jak již je zmíněno výše, používá balík `JFreeChart`. Třída tvoří nezbytná nastavení pro vykreslení grafu a obsahuje kromě konstruktoru dvě metody. Metoda `prepareDataset()` má za úkol při-

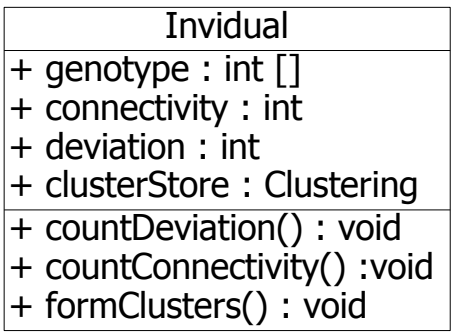


Obrázek 4.1: Schéma třídy Chart

pravit dataset, který převezme jako argument při volání a je typu `XYDataset`. Následně připraví okno, ve kterém se graf vykreslí. Pro vykreslení grafu je použita druhá metoda `createChart()`. V této metodě se již nastavují nezbytné proměnné pro správné vykreslení grafu jako např. barvy, osy, popisky apod.

4.2 Třída Individual

Třída Invidual vytváří jedince, tedy samostatné řešení. Handl a Knowles poměrně přesně specifikují, co by každé řešení, tedy tato třída, mělo obsahovat.



Obrázek 4.2: Schéma třídy Invidual

Přesně popsaná je proměnná `genotype`. Jedná se o pole typu `int[NodesCount]`, tedy pole o velikosti počtu vrcholů. Grafová reprezentace byla popsaná v části 2.1.5.

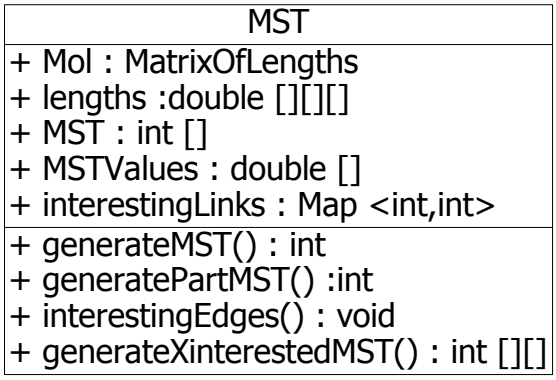
Nyní stačí zopakovat pouze to, že takto lze uložit graf pouze díky tomu, že je neorientovaný a také díky tomu, že každý shluk je ukládán jako minimální kostra, tudíž zde stačí pouze tolik hran, kolik je vrcholů.

Dále třída obsahuje proměnné metrik: `connectivity` a `deviation`. Tyto metriky ohodnocují řešení pomocí patřičných metod `countConnectivity()` a `countDeviation()`. Třída také samozřejmě implementuje gettery a settery pro výše zmíněné proměnné.

Poslední proměnnou je `clusterStore`, která je typu *Clustering*. Interface Clustering je implementované ve frameworku Clueminer. Proměnná má svou metodu `formClusters()`, která pro uložený genotyp vytvoří shlukové řešení, které je potřeba pro hodnocení (výpočet metrik) řešení.

4.3 Třída MST

Tato třída[11] slouží pro hledání minimální kostry v grafu. Tuto třída se použije v inicializační části programu, kde je potřeba nalézt MST řešení. K nalezení kostry je použit Jarníkův [19] algoritmus, který je však spíše známý pod jménem Roberta Prima jako Primův [16]. K nalezení minimální kostry (metoda



Obrázek 4.3: Schéma třídy MST

`generateMST`()) potřebuje algoritmus vědět vzdálenosti mezi jednotlivými vrcholy, k tomu si algoritmus předvypočítá matici vzdáleností, která se při volání konstruktoru uloží do proměnné *MoL*. Samotná matice vzdáleností bohužel nestačí, je potřeba mít tyto hodnoty seřazené, tedy mít pro každý vrchol seznam jeho nejbližších vrcholu vzestupně seřazených. Seznam se získá zavoláním metody `MoL.getSortedRow(i)`, která vrací pro daný (i vrchol vrátí seřazený seznam nejbližších vrcholů včetně vzdáleností. Tuto informaci si algoritmus uloží do proměnné `lengths`. Dále v této metodě následuje Jarníkův/Primův

[11] MST zkratka pro minimal spanning tree – angl. minimální kostra grafu

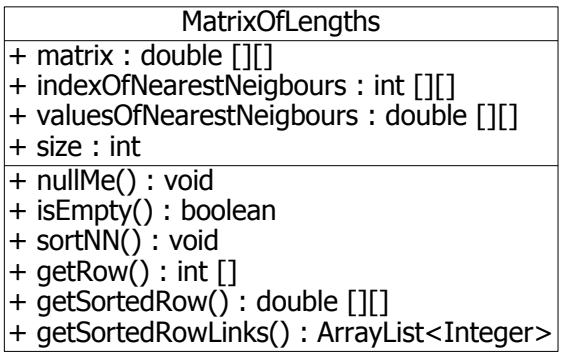
algoritmus, který nalezne kostru a její *genotyp* uloží do proměnné `MST` a vzdálenosti jednotlivých hran do `MSTValues`.

Další důležitou částí algoritmu je dvojice metod `interestingEdges`() a `generateXinterestedMST()`. První metoda hledá v kostře všechny zajímavé hrany. Druhá metoda pak generuje MST modifikovaná řešení, kde jsou zajímavé hrany zmutovány a vedou do bližších vrcholů. Co přesně jsou zajímavé hrany a jak se generují zajímavá MST řešení, je popsáno v kapitole MOCK (2), konkrétně v části Generování zajímavých MST řešení (viz. 2.1.8).

Poslední metodou této třídy je `generatePartMST()`. Tato metoda není spojena přímo s tvorbou minimální kostry, ale jak již jméno napovídá, s generováním kostry pouze pro části grafu. Tuto metodu algoritmus využívá při hledání k-Means řešení, když potřebuje zjistit vnitřní strukturu shluku. Mohlo by se zdát, že struktura shluku není důležitá při hledání těchto řešení, také že není. Struktura je však důležitá pro následuji část algoritmu, kdy řešení budou mutovat.

4.4 Třída Matrix Of Lengths

Třída *MoL* slouží k ukládání vzdáleností mezi vrcholy. Tyto vzdálenosti se ukládají do proměnné `matrix`. Dále je zde uložen seznam indexů *indexOfNearestNeighbours* a hodnot *valuesOfNearestNeigbours*. Tyto dvě pole je nejprve potřeba spočítat, a to algoritmus činí pomocí metody `sortNN()`. Dále třída

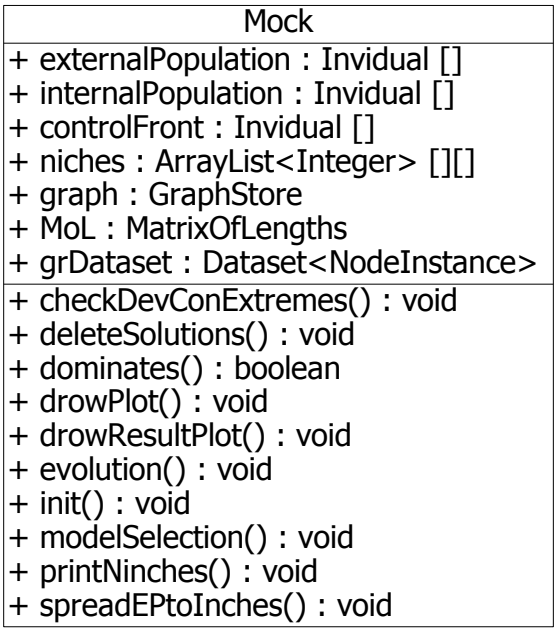


Obrázek 4.4: Schéma třídy *MoL*

obsahuje funkci pro vyčištění matice `nullMe()` a boolean `isEmpty()` pro zjištění, zda je matice prázdná. Metoda `getSortedRow()` vrací seřazený seznam nejbližších vrcholů, tato metoda je volaná v rámci třídění matice v metodě `sortNN()`. Dále již třída obsahuje pouze klasické gettery a settery.

4.5 Třída Mock

Tato třída představuje hlavní část algoritmu MOCK. Na rozdíl od ostatních tříd je poměrně rozsáhlá, obsahuje mnoho proměnných, z nichž část je pevně nastavena, jedná se o parametry algoritmu, které jsou popsané již výše v části 2.3. Kvůli složité struktuře a množství proměnných bude tato třída po-



Obrázek 4.5: Schéma třídy Mock

psaná chronologicky, tedy podle běhu algoritmu. Algoritmus začíná zavoláním funkce `init()`. Tato funkce slouží pro inicializaci algoritmu MOCK. Nejprve se načte dataset do GraphStoru, jednotlivé parametry se uloží do ColumnStore, což je uzpůsobený kontejner pro parametry vrcholů, implementovaný v rámci GraphStore. Dále je potřeba spočítat matici vzdáleností (`MoL`) a seřadit ji.

Nyní je již nejnutnější inicializace ukončená a algoritmus začne hledat minimální kostru grafu za použití metody `generateMST()` v třídě *MST*. Následuje hledání zajímavých hran a generování zajímavých MST řešení.[12] Nyní algoritmus již ví, kolik je zajímavých hran, a dopočítá, jak bude velké k při běhu k-Means. Jednotlivá řešení se získají použitím metody `giveKmeans()` ze třídy kMeans. Získaná řešení algoritmus uloží a tímto končí inicializace. Algoritmus našel `fsize` řešení, přičemž platí, že první část, avšak maximálně polovina, je z MST zajímavých řešení, zbytek jsou k-Means řešení.

[12] angl. *generating n MST solution with variable interesting edges modified*

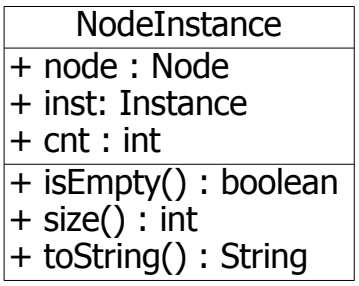
Algoritmus pokračuje funkcí `evolution()`, kde probíhá evoluční část algoritmu. Nejprve je potřeba připravit pro evoluci několik proměnných, inicializovat `niches`, zkopírovat si hlubokou kopií inicializační řešení do `controlFront`, spočítat extrémy *connectivity* a *deviation* pomocí metody `checkDevConExtremes()` a rozprostřít řešení do patřičného *niche* pomocí metody `spreadEPtoInches()`. Nyní je již vše připraveno pro vlastní evoluci.

Evoluce iteruje celkem *numOfGenerations*-krát. Na začátku každé iterace nejdříve algoritmus vybere z `niches` 10 náhodných řešení (interní populace), přičemž se snaží je vybrat napříč celým spektrem. Každé řešení z interní populace algoritmus nechá zkřížit (viz část 2.1.6) a zmutovat (viz část 2.1.7). Algoritmus dále najde shluková řešení pro nová řešení, spočítá jim *connectivity* a *deviation*. V případě, že je již příliš mnoho jedinců v externí populaci, tak ji promaže. Nakonec přidá nová řešení do externí populace, pokud dominují. Pokud některé z nových řešení má větší či menší hodnotu než patřičný extrém *connectivity* a *deviation*, tak algoritmus přepočítá všechny extrémy, normalizuje je a přerozdělí je znovu v proměnné *niches*.

Poslední částí algoritmu je funkce `modelSelection()`, kde se vybírá nejlepší řešení. Tato funkce je popsaná v části: Modelová část viz. 2.2, Handl a Knowles tuto část popisují velice podrobně pomocí pseudokódu.

4.6 Třída Node Instance

Tato pomocná třída implementuje `Instance<Double>` a slouží pro práci s instancemi vrcholů při tvorbě k-Means. Obsahuje mnoho pomocných metod,

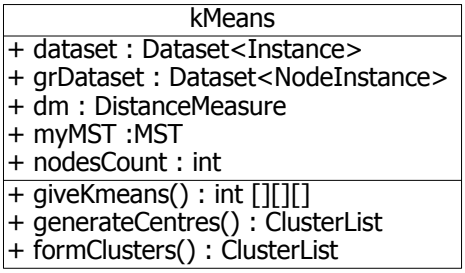


Obrázek 4.6: Schéma třídy Node Instance

většina z nich však pro běh algoritmu není potřeba, proto jsou implementovány jen základní metody pro práci s proměnnými a zbylé metody jsou v režimu *Not supported yet*. Implementované metody jsou jednoduché, jejich smysl je jasný již z názvu a jejich implementace je jednoduchá, není k ní potřeba komentář.

4.7 Třída *k*-Means

Třída k-Means slouží nalezení k různých řešení pomocí stejnojmenného algoritmu [20]. Třída má jednu hlavní metodu `giveKmeans()`, která generuje všecha řešení a vrací je uložené v třírozměrném poli typu *int*. Metoda na začátku každé iterace k, která probíhá v intervalu $< 2, n >$, nejprve získá z metody `generateCentres()` jednotlivá centra budoucích shluků. Následuje



Obrázek 4.7: Schéma třídy k-Means

vytvoření shluků a 10 iterací, během kterých se přepočítávají a formují shluky tak, aby konečné shlukové řešení bylo co nejvíce spravedlivé. Tyto dvě metody však jsou navržené a implementované způsobem, že pro její běh potřebuje algoritmus Dataset<Instance> `dataset` a Dataset<NodeInstance> `grDataset`. Následuje hledání vnitřního schéma každého shluku, pomocí výše zmíněné metody `MST.generatePartMST()`. Poté algoritmus uloží získaný genotyp shluku a daná iterace končí.

Tato třída generuje druhou část počátečních řešení. Nutno dodat, že obě části nemusí být stejně velké, protože velikost k přímo závisí na počtu nalezených zajímavých hran.

Kapitola 5

Popis metodiky testování a jeho výsledky

Tato kapitola popisuje průběh testování algoritmu. Nejdříve se testovala správná funkčnost samotných metod programu. Nebyly implementovány testovací třídy, protože v tomto případě se testování jednotlivých modulů při práci s grafy vyhodnocuje vizuálně. Správná funkcionalita elementárních metod byla otestována na datasetu IRIS[13].

5.1 Testování funkcionality

Testování správné funkcionality třídy MST a k-Means bylo testováno programem Graphviz[14]. Testování spočívalo ve výpisu genotypu jednotlivých řešení a následné zobrazení grafu v Graphvizu. Graphviz zobrazuje pouze strukturu, nikoli fyzické rozprostření bodů, proto testování bylo ještě doplněno porovnáním struktury z graphvizu s jednotlivými vrcholy a jeho souřadnicemi v textovém souboru. V rámci datasetu IRIS se nejednalo o příliš velký problém, ovšem v případě testování většího datasetů by to již problém mohl být.

Tímto způsobem byly testovány metody `generateMST()`, `generate-PartMST()` a `generateXinterestedMST()` v rámci třídy MST. V rámci třídy k-Means takto byly testovány metody `giveKmeans()` a `formClusters()`. Správné rozmístění shluků bylo také testováno pomocí metody `drowPlot()`, která vykresluje grafové zobrazení pomocí balíčku frameworku JFreeChart[15]. Z grafu následně bylo vizuálně ověřena správnost shlukového řešení pomocí obarvení jednotlivých shluků. Bylo tedy ověřeno, že inicializační část programu generuje správná řešení, protože generuje řešení pouze z výše zmíněných metod.

[13] `https://archive.ics.uci.edu/ml/datasets/Iris`
[14] `http://www.graphviz.org/`
[15] `http://www.jfree.org/jfreechart/`

Grafové zobrazení generované frameworkem JFreeChart však není příliš přehledné, proto jsou níže grafy generované programem Gnuplot.[16]

Testování evoluční části probíhalo v několika krocích. Nejdříve byla testovaná mutace a křížení, kdy se opět pomocí programu Graphviz testovalo, zda řešení mutují a kříží se správně. Protože mutace a křížení probíhá pouze na hranách, tak při jejich testování ani nebylo podstatné zkoumat fyzické souřadnice vrcholů. Dále se testovalo správné přidávaní nových řešení. Ověřovalo se, aby se do externí populace nepřidávala duplicitní řešení či řešení téměř identická. Za téměř identická řešení lze považovat taková, jenž mají shodné shlukové řešení, stejnou hodnotu hodnotících funkcí *connectivity* a *deviation*, taková řešení se liší pouze vnitřní strukturou uvnitř shluku, kdy mají pouze např. jeden prvek na okraji shluku připojený přes jiný prvek. Důležité testování správného generování *Pareto* fronty probíhalo kontrolou správného generování vektorů při PCA, generování datasetu a jednotlivých řešení, fronta dále byla vykreslena metodou `drowResultPlot()`. Tato metoda vytváří, podobně jako metoda `drowPlot()` za pomocí frameworku JFreeChart, graf, na kterém je *Control* fronta a momentální stav externí populace. Změny v rámci externí populace je také možné kontrolovat metodou `printNiches()`, která umožňuje pozorovat aktuální stav proměnné `niches`.

Ověřování správné funkcionality Modelové části bylo elementární ze dvou důvodů. Prvním důvodem je fakt, že algoritmus v této části poměrně je jednoduchý, nic složitého se v ní nepočítá. Druhým důvodem je skutečnost, že tuto část popisují Handl a spol.[4](strana 10) pomocí pseudokódu. Správnost metody byla otestována pomocí výpisu všech nejbližších vzdáleností, ze kterých bylo správně nalezeno nejvzdálenější řešení od *Control* fronty.

5.2 Ověřování správností výsledků

V této části je popsána metodika ověřování právnosti výsledků. Algoritmus byl testovaný kromě výše zmíněného datasetu IRIS také na datasetech WINE[17], GLASS[18], SONAR[19] a VEHICLE[20]. Na těchto datech byl testován algoritmus v běhu na 10 000 generací. Z grafu 5.1 je patrný vývoj počtu nově nalezených dominujících řešení v průběhu generací evoluce. Počet dominujících řešení zde ještě není snížen optimalizací, kdy budou vyndány z *Pareto* fronty všechna řešení, kterým dominuje nějaké jiné řešení *Pareto* fronty. Všechna nalezená řešení dominující řešení dominují ve svém ninchi v době nalezení. Tato optimalizace implementace v textu Handla a spol. implicitně není popsaná, ale

[16] `http://www.gnuplot.info/`

[17] `https://archive.ics.uci.edu/ml/datasets/Wine`

[18] `https://archive.ics.uci.edu/ml/datasets/Glass+Identification`

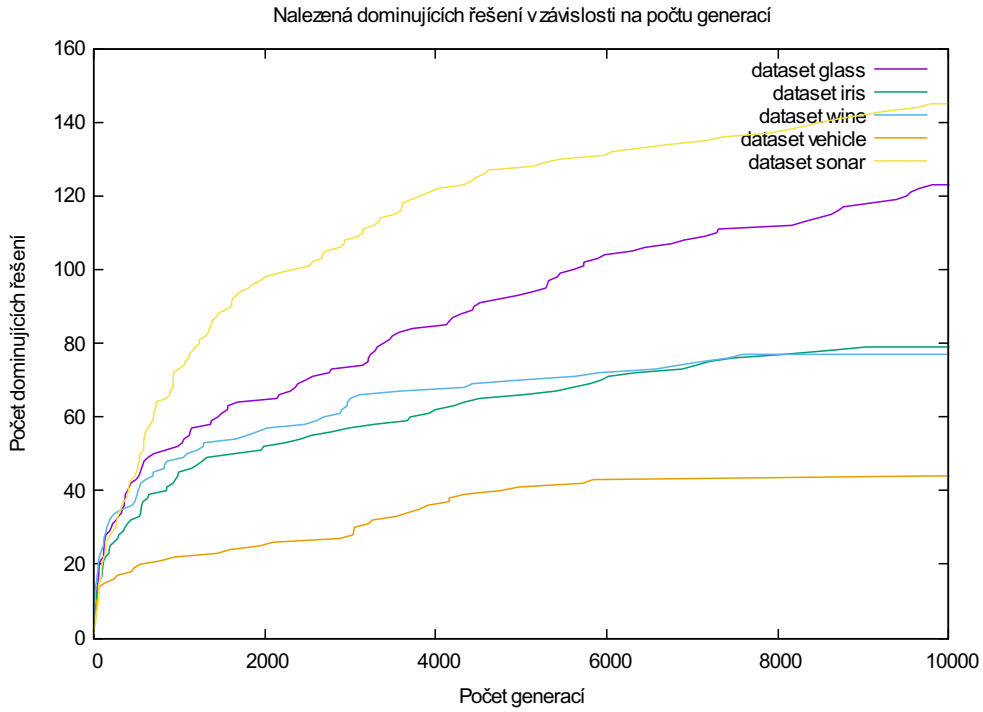
[19] `https://archive.ics.uci.edu/ml/datasets/Connectionist+Bench+%28Sonar,+Mines+vs.+Rocks%29`

[20] `https://archive.ics.uci.edu/ml/datasets/Statlog+%28Vehicle+Silhouettes%29`

Jméno	Počet instancí	Počet atributů	Doba běhu na 10 000 generací
Glass	214	10	4:13 min.
Iris	150	4	2:38 min.
Sonar	208	60	7:10 min.
Vehicle	946	18	85 min.
Wine	173	13	4:20 min.

Tabulka 5.1: Porovnání datasetů

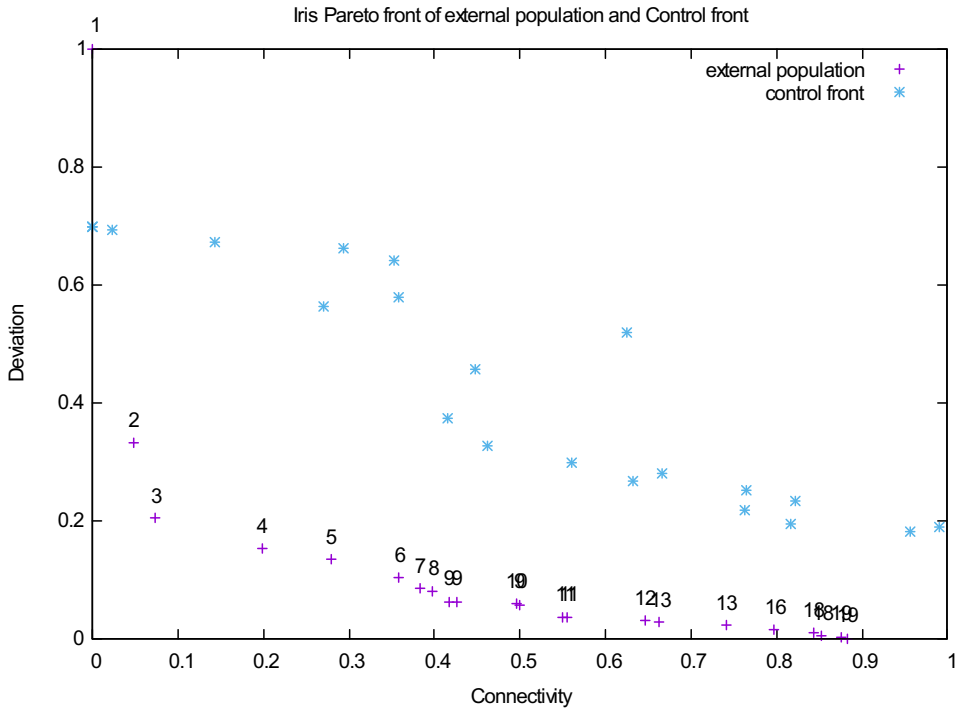
očekává se. Pro všechny testované datasety platí, že podstatná část dominujících řešení byla nalezena v prvních 1000 generacích. Toto měření potvrzuje slova Handl a spol a jejich zvolení hodnoty 1000 pro počet generací, kdy algoritmus při běhu s větším počtem generací nachází méně řešení, navíc čím dál tím podobnější. Na obrázku 5.2 jsou zobrazená výsledná řešení pro běh



Obrázek 5.1: Vývoj počtu nalezených dominujících řešení

algoritmu na datasetu IRIS v implicitním nastavení. Bylo zde nastaveno maximální počet shluků stejně jako počáteční počet řešení fsize na hodnotu 25, protože dataset IRIS obsahuje pouze 1 zajímavou hranu a tudíž nemá smysl

generovat zbytečně vysoký počet k-Means řešení.

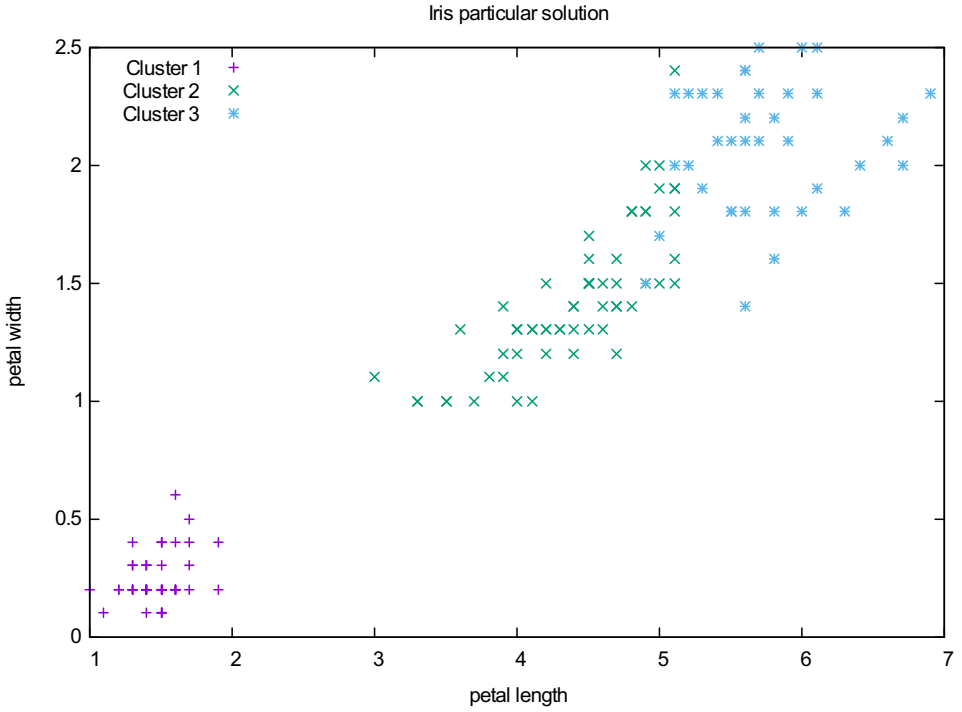


Obrázek 5.2: Výsledná *Pareto* fronta datasetu IRIS

Pareto fronta byla z optimalizována, když z ní byla smazána ta řešení, kterým dominuje jiný prvek *Pareto* fronty. Popisky u jednotlivých řešení *Pareto* fronty externí populace značí počet shluků, které má dané řešení. Z grafu je patrné, že nejvzdálenější od *Control* fronty je řešení se 3 shluky. Řešení je vykreslené na obrázku 5.3.

5.2.1 Výsledná řešení

Výsledkem algoritmu MOCK není jedno řešení, nýbrž sada řešení. Tato sada obsahuje všechna dominující řešení, v případě velkého počtu dominujících řešení lze ještě vyfiltrovat podobná řešení tak, aby se zahodila velmi podobná řešení. Výsledná sada řešení se nazývá *Pareto optimal* fronta a získá se zahozením řešení, kterým dominuje jiné řešení externí populace. Výsledný výběr nejlepšího řešení z *Pareto* fronty velmi ovlivňuje, jak se podaří algoritmu vygenerovat data a ohodnotit jejich řešení v rámci *Control* fronty, protože právě vzdálenost mezi nejbližším prvek *Control* fronty přímo ovlivňuje nejlepší výsledek. Handl a spol. se občas sami potýkají s problémy během generování



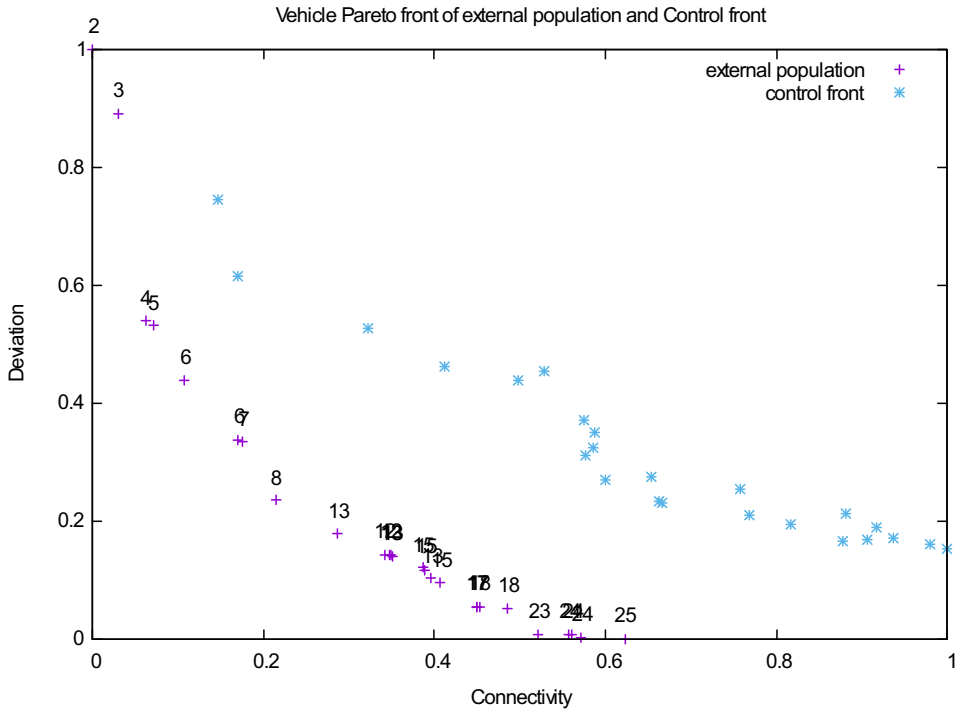
Obrázek 5.3: Výsledné shlukové řešení *Pareto* fronty datasetu IRIS

dat do *Control* fronty, konkrétně s PCA [21]. I z tohoto důvodů je proto lepší uvádět jako výsledek celou frontu místo jednoho konkrétního řešení.

5.2.2 Porovnání výsledků s ostatními algoritmy

Výsledky algoritmu MOCK (viz. 5.4) se dle zadání mají porovnat s tradičními přístupy ke shlukování (k-Means a hierarchické shlukování), dále pak s algoritmy MCL (Markov Cluster Algorithm [22]) a CW (Chinese Whispers [23]). Metody k-Means a metody hierarchického shlukování jsou jsou již popsány v kapitole 1. Algoritmus k-Means je obsažený v rámci inicializační části algoritmu MOCK, proto nemá příliš logiku ho zde porovnávat sámotný. Jediný rozdíl v implementaci v tomto algoritmu oproti běžně používané implementaci je v omezení přepočítávání středů shluků na L cyklů. Algoritmy hierarchického shlukování (nejčastěji Average link a single link) obecně dosahují horších výsledků, než algoritmy inteligentnější, to potvrzuje i měření Handl a spol. Algoritmus MCL je rychlý a škálovatelný shlukovací algoritmus bez učitele pro grafy, je založený na simulaci stochastických proudů v grafu.[21] Algoritmus Chinese Whispers je také grafový algoritmus, který se značí velkou rych-

[21] http://micans.org/mcl/



Obrázek 5.4: Výsledná *Pareto* Fronta datasetu VEHICLE

lostí (časově lineární v závislosti na počtu hran – v případě MOCKu n^2 hran) a generuje neomezený počet shluků. Jeho nevýhodou podobně jako jakýkoli evoluční algoritmus, tedy i MOCK, je nedeterminovanost.

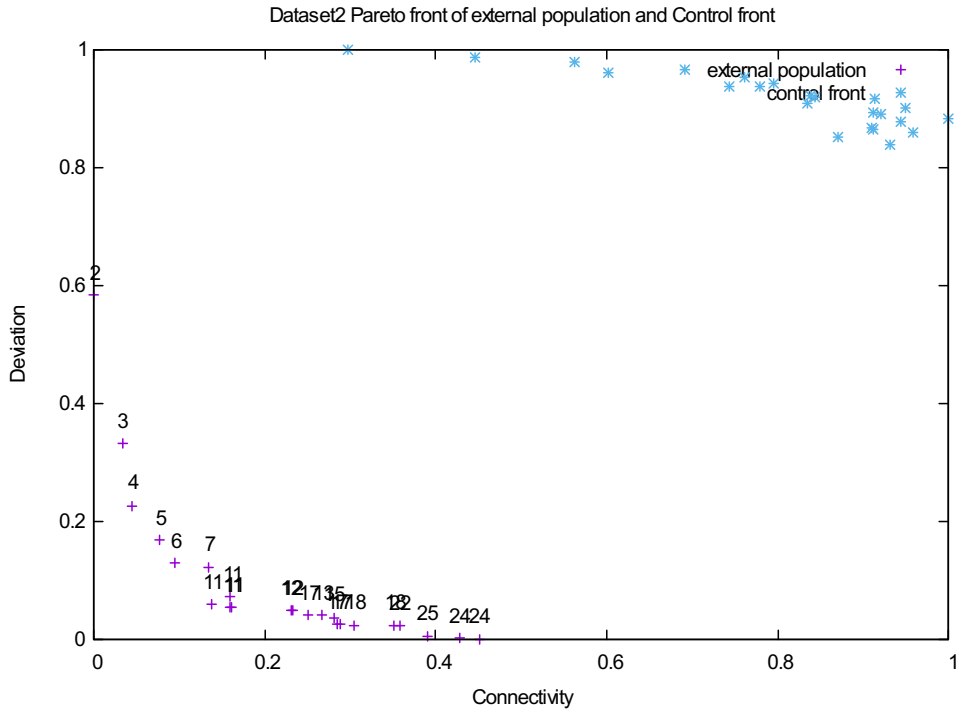
5.2.3 Testování na reálných datech

Nasazením algoritmu na náhodná data se plně využívá největší přednost algoritmu MOCK. Na reálných datech už zpravidla nefungují lépe algoritmy založené na hustotě, které dosahují dobrých výsledků kvůli způsobu získání – generování dat. Největší předností algoritmu MOCK je komplexnost výsledných *Pareto optimal* řešení, která obsahují nejlepší řešení napříč spektrem hodnotících funkcí. Právě tato vlastnost dává algoritmu výhodu před ostatními algoritmy, které vrací pouze jedno řešení.

Výsledky algoritmu na reálných datech ukázaly podobné problémy s generováním *Control* fronty popsané např. v [21]. Výsledný graf datasetu DNA microarray[22] *Pareto* fronty a *Control* fronty je na obrázku 5.5. U jiných datasetů s větším počtem instancí se někdy objevila chyba, kdy program spadl pro nedostatečnou velikost paměti. Tento problém by do se budoucna jistě dal

[22] `http://en.wikipedia.org/wiki/DNA_microarray`

odladit zjednodušením struktury a tedy menší datovou vytížeností. Znovu se naráží na problematiku generování *Control* fronty, kvůli které se musí vygenerovat druhý stejně velký dataset a spolu s jinými strukturami jako matice vzdáleností apod. to může být paměťově velice náročné.



Obrázek 5.5: Výsledná *Pareto* Fronta na datasetu DNA microarray

Závěr

Tato bakalářská práce vznikla na základě potřeby implementovat evoluční algoritmus MOCK do frameworku Clueminer. Cílem práce bylo seznámení se s různými technikami shlukování dat, funkcionalitou frameworku Clueminer, podrobné nastudování algoritmu MOCK, tak jak ho popisuje Handl a Knowles [4] a jeho následná implementace do frameworku Clueminer.

Implementoval jsem shlukovací evoluční algoritmus MOCK, tak jak byl popsán autory a porovnal s ostatními algoritmy, cíle práce tedy byly splněny. Algoritmus je poměrně složitý a výpočetně náročný pro velký počet instancí, nicméně jeho uplatnění vidím např. v testování reálných dat s menším počtem instancí a větším počtem atributů, kde nás zajímají všechny možné souvislosti (shluky), které algoritmus vypočítá v *Pareto* frontě.

Současná implementace obsahuje řešení, které by se dalo z optimalizovat v několika směrech. Implementace neobsahuje více-vláknovou implementaci a některé algoritmy nepoužívají nejefektivnější řešení. Rychlost algoritmu v současné podobě je poměrně nezávislá na počtu atributů, protože s nimi počítá pouze při výpočtu matice vzdáleností a v PCA. Větší počet instancí ovšem velmi zpomaluje rychlost algoritmu a zvyšuje také velmi paměťovou složitost.

Literatura

[1] Deza, M. M.; Deza, E.: *Encyclopedia of Distances*. Springer Berlin Heidelberg, 2009, doi:10.1007/978-3-642-00234-2_1.

[2] Nesetril, J.; Milková, E.; Nesetrilová, H.: Otakar Boruvka on minimum spanning tree problem Translation of both the 1926 papers, comments, history. *Discrete Mathematics*, ročník 233, č. 1-3, 2001: s. 3–36.

[3] MacQueen, J. B.: Some Methods for Classification and Analysis of MultiVariate Observations. In *Proc. of the fifth Berkeley Symposium on Mathematical Statistics and Probability*, ročník 1, editace L. M. L. Cam; J. Neyman, University of California Press, 1967, s. 281–297. Dostupné z: `http://www.bibsonomy.org/bibtex/25dcdb8cd9fba78e0e791af619d61d66d/enitsirhc`

[4] Handl, J.; Knowles, J.: An Evolutionary Approach to Multiobjective Clustering. *IEEE Transactions on Evolutionary Computation*, ročník 11, č. 1, 2007: s. 56–76. Dostupné z: `http://www.bibsonomy.org/bibtex/242c4c9555bfd2f71a4edc0557eaaa978/dalbem`

[5] Ward, J.: Hierarchical grouping to optimize an objective function. *Journal of the American Statistical Association*, ročník 58, 1963: s. 236–244. Dostupné z: `http://www.bibsonomy.org/bibtex/2508321626aa8044fe77e0d1e2e234af4/gromgull`

[6] Tan, P.; Steinbach, M.; Kumar, V.: *Introduction to Data Mining*. Addison Wesley, první vydání, Květen 2005, ISBN 0321321367.

[7] Defays, D.: An efficient algorithm for a complete link method. *The Computer Journal*, ročník 20, č. 4, 1977: s. 364–366.

[8] Sibson, R.: SLINK: an optimally efficient algorithm for the single-link cluster method. *The Computer Journal*, ročník 16, č. 1, 1973: s. 30–34.

[9] Kruskal, J. B.: On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem. In *Proceedings of the American Mathematical Society, 7*, 1956.

[10] Sokal, R. R.; Michener, C. D.: A statistical method for evaluating systematic relationships. *University of Kansas Science Bulletin*, ročník 38, 1958: s. 1409–1438.

[11] Dodge, Y.: *Statistical data analysis based on the L1-norm and related methods*. Birkhauser, 2002, ISBN 3764369205.

[12] Ng, R.; Han, J.: Efficient and effective clustering methods for spatial data mining. In *Proceedings of the 20th international conference on Very Large Data Bases (VLDB'94)*, Morgan Kaufmann, September 1994, s. 144–155.

[13] Ester, M.; Kriegel, H.-P.; Sander, J.; aj.: A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *Proc. of 2nd International Conference on Knowledge Discovery and*, 1996, s. 226–231.

[14] Corne, D.; Jerram, N.; Knowles, J.; aj.: PESA-II: region-based selection in evolutionary multiobjective optimization. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, 2001.

[15] Jolliffe, I. T.: *Principal Component Analysis*. Springer, 2002, ISBN 0387954422.

[16] Prim, R. C.: Shortest Connection Networks and some Generalizations. *The Bell Systems Technical Journal*, ročník 36, č. 6, 1957: s. 1389–1401.

[17] Barton, T.: *Cluster analysis of cell profile responses*. Diplomová práce, Czech Technical University, 2011.

[18] Fowler, M.: Reducing Coupling. *IEEE Software*, ročník 18, č. 4, 2001: s. 102–104.

[19] Milková, E.: The Minimum Spanning Tree Problem: Jarník's solution in historical and present context. *Electronic Notes in Discrete Mathematics*, ročník 28, 2007: s. 309–316.

[20] Hartigan, J.: *Clustering Algorithms*. John Wiley and Sons, New York, 1975. Dostupné z: `http://www.bibsonomy.org/bibtex/21e578b491641715a807d1b6b136d2204/danielt`

[21] Faceli, K.; de Souto, M. C. P.; de Araujo, D. S. A.; aj.: Multi-objective clustering ensemble for gene expression data analysis. *Neurocomputing*, ročník 72, č. 13-15, 2009: s. 2763–2774.

[22] van Dongen, S.: *Graph Clustering by Flow Simulation.* Dizertační práce, University of Utrecht, 2000.

[23] Biemann, C.: Chinese Whispers - an Efficient Graph Clustering Algorithm and its Application to Natural Language Processing Problems. 2006, s. 73–80.

Příloha **A**

Seznam použitých zkratek

CLARANS – Clustering Algorithm based on Randomized Search

CLINK – Complete-linkage clustering

CW – Chinese whispers

DBSCAN – Density-based spatial clustering of applications with noise

MCL – Markov Cluster Algorithm

MOCK – Multiobjective clustering with automatic k-determination

MST – Minimal spanning tree

SLINK – Single-linkage clustering

UPGMA – Unweighted pair group method with arithmetic mean

Příloha **B**

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `jar` ..... adresář se spustitelnou formou implementace
- `src`
  - `impl` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF